COMPUTING SCIENCE

Event-B Day. National Institute of Informatics
Tokyo, Japan. November 21, 2016.

Fuyuki Ishikawa, Alexander Romanovsky.

TECHNICAL REPORT SERIES

No. CS-TR-1504 November 2016

TECHNICAL REPORT SERIES

No. CS-TR-1504 November, 2016

Event-B Day. National Institute of Informatics
Tokyo, Japan. November 21, 2016.

Fuyuki Ishikawa, Alexander Romanovsky.

Abstract

Event-B is a formal method for the system level modelling and analysis of dependable applications. It is supported by an open and extendable Eclipse-based toolset called Rodin, which has been developed in a series of European projects. This one day event aims to bring the community of Event-B/Rodin users and developers together to discuss new and emerging issues in applying and advancing both the Event-B method and the Rodin platform as well as address challenges that industrial takers are facing while deploying them.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1504

Abstract

Event-B is a formal method for the system level modelling and analysis of dependable applications. It is supported by an open and extendable Eclipse-based toolset called Rodin, which has been developed in a series of European projects. This one day event aims to bring the community of Event-B/Rodin users and developers together to discuss new and emerging issues in applying and advancing both the Event-B method and the Rodin platform as well as address challenges that industrial takers are facing while deploying them.

About the authors

Fuyuki Ishikawa is an Associate Professor at the National Institute of Informatics (NII) in Tokyo, Japan. He received the PhD degree in information science and technology from the University of Tokyo in Japan, in 2007. He is a visiting associate professor at the University of Electro-Communications in Japan. His research interests include service-oriented computing and software engineering. He has served as the leader of 6 funded projects and published more than 100 papers.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. He is the leader of the Dependability Group at the School of Computing Science. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance.He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, University of Newcastle upon Tyne. Alexander is now the

Principle Investigator of the TrAmS-2 EPSRC/UK platform grant on Trustworthy Ambient Systems (2012-16) and of the EPSRC/RSSB research project SafeCap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety (2011-14), and the Co-investigator of the EPSRC PRiME program grant (2013-18) and of the FP7 COMPASS Integrated Project (2011-14).

Suggested keywords

Formal methods, Event-B, formal verification, refinement, Rodin

Event-B Day

November 21, 2016

National Institute of Informatics
Tokyo, Japan

Organisers:

Fuyuki Ishikawa (NII, Japan)
Alexander Romanovsky (Newcastle University, UK)

Welcome to the Event-B Day in Tokyo!

Event-B is a formal method for the system level modelling and analysis of dependable applications. It is supported by an open and extendable Eclipse-based toolset called Rodin, which has been developed in a series of European projects.

This one day event aims to bring the community of Event-B/Rodin users and developers together to discuss new and emerging issues in applying and advancing both the Event-B method and the Rodin platform, as well as to address challenges that industrial takers are facing while deploying them.

The programme of the event consists of twelve presentations to be given by the Event-B experts from all over the world.

We hope you will enjoy the event.

Fuyuki Ishikawa and Alexander Roemanovsky

Programme

8.30-Coffee

8.55-9.00. Welcome from the organisers
9.00-9.30. Jean-Raymond Abrial (France). Formal proof of the Weak Goodstein Theorem.
9.30-10.00. Yamine Ait Ameur (IRIT / INPT-ENSEEIHT, France). Correct Refinement of Continuous Models
10.00-10.30. Hironobu Kuruma (Hitachi, Ltd., Japan). Crossed-Project Reference for Modular Modeling

10.30-10.50. Coffee

10.50-11.20. Marc Frappier (Université de Sherbrooke, Canada). Generating Vulnerability Tests for Java Card Structural Constraints
11.20-11.50. J Paul Gibson (Telecom Sud Paris, France). When Students Choose to Use Event-B in their Software Engineering Projects
11.50-12.20. Alexei Iliasov (Newcastle University, UK). Verifying Paxos protocol in Event-B

12.20-13.20. Lunch

13.20-13.50. Fuyuki Ishikawa (NII, Japan). Event-B Refinement Restructuring and its Applications
13.50-14.20. Thai Son Hoang (University of Southampton, UK). Theory Plug-in for Rodin 3.x
14.20-14.50. Filali Amine (IRIT, France). Modeling and Verifying Event-B Transformations in Event-B

14.50-15.10. Coffee

15.10-15.40. Elena Troubitsyna (Abo Akademi, Finland). Towards Security-Explicit Formal Modelling of Safety-Critical Systems
15.40-16.10. Paulius Stankaitis (Newcastle University, UK). Automating Verification of Event-B Models
16.10-16.40. Rajiv Murali (Heriot-Watt University, UK). A Formal Approach to Use Case Driven Testing.
16.40-Closing discussion. Wrap-up

Abstracts and papers:

Jean-Raymond Abrial. Formal proof of the Weak Goodstein Theorem.

Hironobu Kuruma and Thai Son Hoang. Crossed-Project Reference for Modular Modeling

J Paul Gibson. When Students Choose to Use Event-B in their Software Engineering Projects

Paulius Stankaitis, Alexei Iliasov, David Adjepon-Yamoah, and Alexander Romanovsky. Automating Verification of Event-B Models

Rajiv Murali, Andrew Ireland, and Gudmund Grov. A Formal Approach to Use Case Driven Testing.

Formal Proof of the Weak Goodstein Theorem

Jean-Raymond Abrial

Marseille, France

1 Motivation

For many years, I have been interested in introducing students to the development of complex systems by means of modelling and refinement. To this end, I did not find anything better than presenting *many examples of system developments*. This is due to my inability to propose a unified theoretical treatment on this matter.

Of course, in these examples, I am always pointing out the importance of using some systematic mathematical approaches. However, I figured out that my examples were not explicit enough on how (mechanical) proofs are performed. So, besides courses presenting these examples and also some courses in various forms of proofs (propositional calculus, first order predicate calculus, set theory), I decided to study the *work of professional mathematicians*, thinking that it could be good examples for students.

I must say that I was a bit disappointed by what I discovered: proofs made by mathematicians, as presented in textbooks, are sometimes (for me) difficult to follow in details and thus could have some bad effects on students. As a consequence, I decided to reconstruct by myself some of the interesting proofs I found in the mathematical literature.

Among the works I already studied and reconstructed are the theorem of Zermelo, the theorem of Cantor-Bernstein, the planar graph theorem of Kuratowski, the topological proof of the infinity of primes of Fürstenberg, the intermediate value theorem of Bolzano, the Archimedean property of the set of Real numbers, and others.

More recently, I found that the Goodstein theorem was also very interesting. The purpose of this short note is to give some information about this theorem and the way I introduce a weak form of it to students.

2 The Goodstein Theorem

The theorem stated and proved in 1944 by Goodstein [1], is quite counterintuitive. To explain why, let me consider a weak form of it called, for this reason, the *weak Goodstein theorem*.

Given a number written in base 2 such as 25, that is 11001 ($2^4 + 2^3 + 1$), we transform it by considering the same notation but this time in base 3, that is $3^4 + 3^3 + 1 = 109$. We then subtract 1, yielding 108. We write now this number in base 4, and subtract 1 again, yielding 319. With base 5, we obtain 717, with base 6, 1423. We continue like this: increasing the base and decreasing the result. As can be seen from what is already mentioned, the successive numbers obtained in

this way seem to grow up very rapidly: 25, 108, 319, 717, 1423, ... Nevertheless, the theorem says that this sequence eventually *decreases and terminates at 0*.

The *strong Goodstein theorem* is a little more general than the weak form what we have just described in the previous paragraph, and it is even more counterintuitive. It is not expressed with the classical base notation, as was the weak Goodstein theorem, but rather with the, so-called, *hereditary base notation* (explained in section 3).

Proofs of these theorems in the literature [1] [2] [3] [4] [5] make use of transfinite ordinal numbers. I found that this approach is rather complicated. So, I am looking for another (simpler) possibility. So far, I partially fail, at least for the strong Goodstein theorem. Its weak form however can be proved in a simple fashion. This is what I present here.

3 Hereditary Base Notation

By using the classical notation in base n (where n is a natural number greater than 1), any natural number a is written as follows (base_n(a)):

$$\text{base\_n}(a) = a_l.n^l + \cdots + a_i.n^i + \cdots + a_0.n^0$$

where all a_i are natural numbers smaller than n. As a simplification for this written form, we omit $0.n^i$, we write $1.n^i$ as n^i , n^1 as n, and n^0 as 1. As an example, 25 (that is 16+8+1) is written as follows in base 2:

$$\text{base\_2}(25) = 2^4 + 2^3 + 1$$

By using a notation in hereditary base n, the exponents i used in the notation in base n are also written in base n and so on (h_base_n(a)):

$$\text{h\_base\_n}(a) = a_l.n^{\text{h\_base\_n}(l)} + \cdots + a_i.n^{\text{h\_base\_n}(i)} + \cdots + a_0.n^0$$

So, all natural numbers appearing when using the hereditary base n notation are smaller than or equal to n. As an example, 25 is written as follows in hereditary base 2:

$$\text{h\_base\_2}(25) = 2^{2^2} + 2^{2+1} + 1$$

4 Data Structures for Base Notations

As we all know, writing a natural number in a certain base consists quite often in removing the base when it is obvious. As a result, we have just a sequence of digits (all smaller than the base). For example, 25 in base 2 is simply written: 11001. Such a sequence is organised as follows: it starts at index 0 and goes from right to left. Each number at index i corresponds to the factor used with exponent i. This is illustrated in Fig.1.

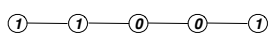

Fig. 1. Sequence representation of $25 = 1.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 1.2^0$

Notice that in this representation we do not omit components of the forms 0.2^i, nor do we omit the 1 in those components of the form 1.2^i. We notice that this sequence representation does not depend on the base: 11001_{10} corresponds to the same sequence (but not the same number) as 11001_2 or 11001_3.

We wonder whether we could do the same (omitting the base) when using the hereditary technique. How could we remove all occurrences of 2 in 25 written in hereditary base 2: $2^{2^2}+2^{2+1}+1$? Here is a more elaborate example: 774840988_{10} in base 3 is $2.3^{18}+3^2+2$ and in hereditary base 3 it is $2.3^{2.3^2}+3^2+1$. *How could we remove all occurrences of 3?*

The idea is to observe that we have three operations in such a representation: addition, exponentiation and multiplication by a factor. The outcome is a *binary tree*, where the horizontal branch corresponds to addition, the vertical branch to exponentiation, and finally the multiplication by a certain factor is just indicated by writing this factor in the corresponding node of the tree. The tree representation of $2.3^{2.3^2}+3^2+1$ (in hereditary base 3) is shown in Fig. 2.

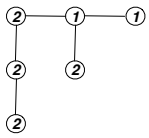

Fig. 2. Tree representation of $2.3^{2.3^2}+3^2+1$ in hereditary base 3

As was the case in the previous section for the sequence representation, it is also very important to notice that the tree representation introduced in this section does not depend on the hereditary base. For example, the number $2.4^{2.4^2}+4^2+1$ is represented by the same tree in hereditary base 4 as is $2.3^{2.3^2}+3^2+1$ in hereditary base 3. To make the distinction between the two, it is necessary to write next to the tree the hereditary base that is used.

In coming sections, I will use the sequence data structure in order to prove the weak Goodstein theorem. I was hoping to use the tree data structure to prove the strong Goodstein theorem. But, so far, I failed.

5 Decreasing Sequence of Natural Numbers

Before engaging in a study of the weak Goodstein theorem in the next section, it is worth considering a simple decreasing sequence of natural numbers. The purpose of this highly simplified case is to show the main mechanism at work, namely lexicographical ordering. It is based on the following simple lemma valid for all positive natural numbers x and n:

$$x^n-1=(x-1).x^{n-1}+(x-1).x^{n-2}+...+(x-1).x^1+(x-1).x^0) \quad \textbf{(Lemma 1)}$$

This lemma is easily provable by induction on n. Applying this lemma to decreasing $24=2^4+2^3$, we obtain the following:

$$\text{Decreasing}(2^4+2^3)=2^4+2^3-1=2^4+(2^3-1)=2^4+2^2+2+1$$

This is illustrated in Fig. 3.

Fig. 3. Sequence representations of $2^4 + 2^3$ and $2^4 + 2^3 - 1$

Likewise, in base 3 we have:

$$\text{Decreasing}(3^4 + 3^3) = 3^4 + 3^3 - 1 = 3^4 + (3^3 - 1) = 3^4 + 2.3^2 + 2.3 + 2$$

This is illustrated in Fig. 4.

Fig. 4. Sequence representations of $3^4 + 3^3$ and $3^4 + 3^3 - 1$

It is interesting to observe the difference between $2^4 + 2^3 - 1$ in Fig. 3 and $3^4 + 3^3 - 1$ in Fig. 4. They both come from the same sequence, either understood to be in base 2 or in base 3. In the second one, all 1 used in the first one are replaced by 2. This is because 2 - 1 = 1 and 3 - 1 = 2.

By decreasing successively an initial number written by means of some base, we obtain a certain sequence and we can prove that such a sequence ends up with the natural number 0. More precisely, we have a *lexicographical ordering*.

6 Informal Proof of the Weak Goodstein Theorem

In the case of weak Goodstein sequences, decreasing is done in the same way as in the previous section except that we increase the base before decreasing. Applying this result to $2^4 + 2^3$, we obtain the following:

$$\text{Weak Goodstein Decreasing}(2^4 + 2^3) = 3^4 + (3^3 - 1) = 3^4 + 2.3^2 + 2.3 + 2$$

Although a weak Goldstein sequence seems to be increasing very rapidly, it happens that such a sequence obtained by applying this process in turn ends up eventually at 0. As a matter of fact, we have the following theorem:

Any weak Goodstein sequence eventually terminates at 0

Informal Proof: We already know that increasing the base does not modify the representation of a number as a sequence. Then decreasing by one after increasing the base just makes the resulting sequence *lexicographically smaller* than the previous one, hence the final result.

Here is the beginning of the weak Goodstein sequence starting at 1000_2:

1000_2 222_3 221_4 220_5 $215_6 \ldots 210_{11}$ $20(11)_{12} \ldots 200_{23}$ $1(23)(23)_{24} \ldots$
$1(23)0_{47}$ $1(22)(47)_{48} \ldots 1(22)0_{95}$ $1(21)(95)_{96} \ldots 1(21)0_{191}$ $1(20)(191)_{192}$
$\ldots 1(20)0_{383}$ $1(19)(383)_{384} \ldots 1(19)0_{767}$ $1(18)(767)_{768} \ldots 1(18)0_{1535} \ldots$

On this sequence, lexicographical decreasing can be seen independently from the current base.

7 A More Formal Treatment of the Weak Goodstein Theorem

7.1 Constructing the sequence $\mathbf{seq_b(n)}$ associated with a number n in base b

$$\mathrm{seq}_b(n) = \begin{cases} \mathrm{seq}_b(n \text{ div } b) \leftarrow n \bmod b & \text{if } \ n \geq b \\ n & \text{if } \ n < b \end{cases}$$

$$\begin{aligned} \mathrm{seq}_2(25) &= \mathrm{seq}_2(12) \leftarrow 1 \\ &= \mathrm{seq}_2(6) \leftarrow 0 \leftarrow 1 \\ &= \mathrm{seq}_2(3) \leftarrow 0 \leftarrow 0 \leftarrow 1 \\ &= \mathrm{seq}_2(1) \leftarrow 1 \leftarrow 0 \leftarrow 0 \leftarrow 1 \\ &= 1 \leftarrow 1 \leftarrow 0 \leftarrow 0 \leftarrow 1 \end{aligned}$$

7.2 Value $\mathbf{val_b(s)}$ of the number associated with a sequence s in base b

$$\begin{aligned} &\mathrm{val}_b(s \leftarrow n) = b.\mathrm{val}_b(s) + n \qquad \text{where } \ n < b \\ &\mathrm{val}_b(n) = n \end{aligned}$$

$$\begin{aligned} \mathrm{val}_2(1 \leftarrow 1 \leftarrow 0 \leftarrow 0 \leftarrow 1) &= 2.\mathrm{val}_2(1 \leftarrow 1 \leftarrow 0 \leftarrow 0) + 1 \\ &= 2^2.\mathrm{val}_2(1 \leftarrow 1 \leftarrow 0) + 0 + 1 \\ &= 2^3.\mathrm{val}_2(1 \leftarrow 1) + 0 + 0 + 1 \\ &= 2^4.\mathrm{val}_2(1) + 2^3 + 0 + 0 + 1 \\ &= 1.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 1.2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

7.3 The Weak Goodstein loop

Next is the loop producing successive elements of the weak Goodstein sequence in the variable n .

$n :=$ some natural number;
$b := 2$;
while $n \neq 0$ **do**
  $n := \mathrm{val}_{b+1}(\mathrm{seq}_b(n)) - 1$;
  $b := b + 1$
end

Now, the question is: *does this loop terminate?*

7.4 Formal Development Outline

Here is the way I develop the course for students. As can be seen, it allows me to develop various formal techniques.

1. How to prove *loop termination*
2. Definitions of *well-founded relations* (demo)
3. Various ways of *proving well-foundedness* (demo)
4. How to prove the *termination of the Weak Goodstein loop*
5. Refresher on *strong well ordering* relations (demo)
6. Refresher of various l*exicographical ordering relations* (demo)
7. *Final Proof* for the Weak Goodstein Loop

References

1. R.L. Goodstein *On the restricted ordinal theorem.* Journal of Symbolic Logic 9(1944)
2. M. Rathjen *Goodstein's Theorem Revisited* Draft (2014)
3. A. E. Caicedo *Goodstein's Theorem.* Revista Columbiana Matematicas (2007)
4. W. Gasarch *Theorems that you simply don't believe.* Computational Complexity blog (2010)
5. L. Kirby and J. Paris *Accessible Independent Results for Peano Arithmetic.* Bulletin of the London Mathematical Society 4 (1982)
6. J. A. Perez *A New Proof of Goodstein Theorem.* Draft (2009)

Crossed-Project Reference for Modular Modeling

Hironobu Kuruma[1] and Thai Son Hoang[2]

[1] Research and Development Group, Hitachi, Ltd., Japan
[2] ECS, University of Southampton, U.K.

Abstract. A typical Event-B model is a collection of components structured by refinement relations and located in a project of RODIN platform. In developing a family of products in industry, it is often observed that a common substructure appears in a variety of models. We experimentally introduced a crossed-project reference mechanism into RODIN platform to investigate the features of modular modeling with component sharing. Providing component reference across projects, it enables to compose a model of components located in several projects. The crossed-project reference modularizes the shared components by classifying them into hierarchical collections and preventing unintended changes. Revising our implementation and making it compatible with other modularization plug-ins such as generic instantiation and theory plug-in is future work.

1 Introduction

In developing a family of products in industry, a variety of models with slight differences are described and verified. A model in Event-B[1] is composed of a collection, typically located within a project, of components combined by **refines**, **extends** or **sees** relations. It is often observed that the same components appear in such a family of models in Event-B. Fig. 1 shows an example family of models containing some common components, i.e., Machine_1, Machine_2, Context_1 and Context_2. Machine_3 and Context_3 (Machine_4 and Context_4) are additional components specific to the model A (the model B respectively).

2 Crossed-Project Reference

We experimentally introduced a crossed-project reference mechanism into RODIN platform[2] to examine modular modeling by means of component sharing. The crossed-project reference manages collections of components and enable references to components between projects so as to share the common components. In Fig.2, the shared components are located in the project C, where the projects A and B contain only the additional components and import the shared components. In each importing project, a **manifest** is used to identify imported components. The names of imported components are prefixed with their source

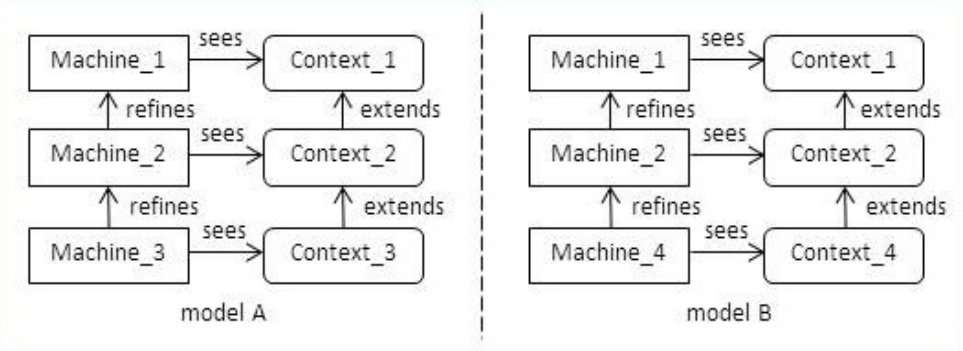

Fig. 1. A Family of Models

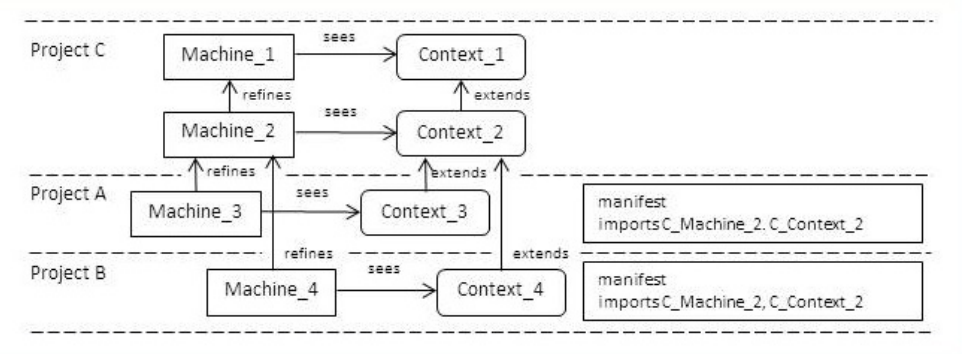

Fig. 2. Crossed-Profect Reference for Component Sharing

project name. The components in importing project refer the imported components by their names declared in the **manifest**. The imports relations between projects must be acyclic.

We implemented the crossed-project reference by renaming and copying the statically checked files of imported components into the importing project. In the above example, two files in the project C, i.e., the statically checked files of Machine_2 and Context_2, are renamed to by prefixing, and copied into the projects A and B. Since the imported components are expected to be verified in their source project, verification is required only for the additional components.

3 Examples

Fig. 3 shows a family of graph structures, rings, strongly connected graphs and tree shaped graphs, defined on the basis of irreflexive transitive closure[1]. They are placed in separate projects and extend a common basic data structure. The irreflexive transition closure is defined by the following two contexts located in the DataStructure project.

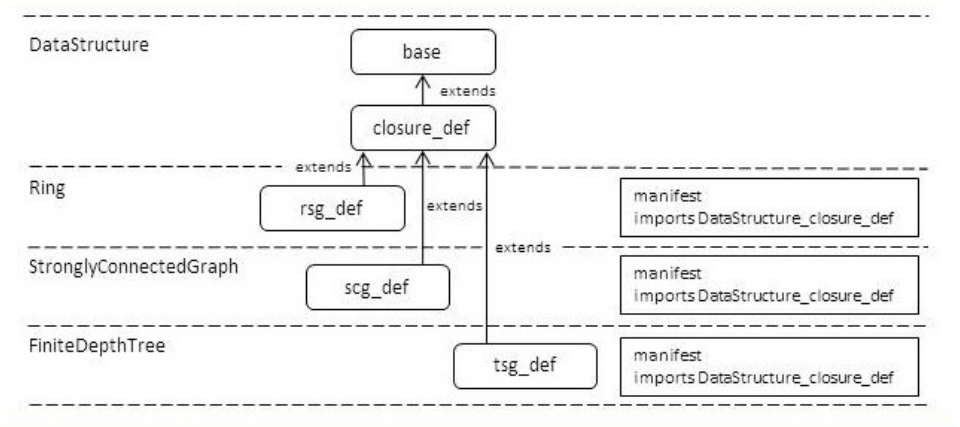

Fig. 3. Family of Graph Structures

base	$\triangleq$	**sets**	S
closure_def	$\triangleq$	**extends**	$base$
		constants	cl
		axioms	$cl = (\lambda r \cdot r \in S \leftrightarrow S \mid (\bigcap f \mid r \subseteq f \wedge f ; f \subseteq f))$

The contexts rsg_def, scg_def and tsg_def refer the closure_def and defines ring shaped graphs, strongly connected graphs and finite depth trees respectively.

rsg_def	$\triangleq$	**extends**	DataStructure_closure_def
		constants	r
		axioms	$\text{finite}(S) \wedge r \in S \rightarrowtail\!\!\!\!\!\twoheadrightarrow S \wedge$
			$\exists l \cdot (l \in S \wedge cl(r)[\{l\}] \cup \{l\} = S)$
scg_def	$\triangleq$	**extends**	DataStructure_closure_def
		constants	r
		axioms	$(S \times S) \setminus \text{id} \subseteq cl(r)$
tsg_def	$\triangleq$	**extends**	DataStructure_closure_def
		constants	$root, leaves, parents$
		axioms	$root \in S \wedge leaves \subseteq S \wedge$
			$parents \in (S \setminus \{root\} \twoheadrightarrow S \setminus leaves) \wedge$
			$cl(parents^{-1})[\{root\}] \cup \{root\} = S$

Fig. 4 shows another example in which a concrete model is derived from the abstract model by assigning values to the carrier sets and constants. The carrier sets and constants of the abstract model are defined abstractly, for example, in the contexts c1 and c2 located in the Abstract project.

c1	$\triangleq$	**sets**	$SEGMENTS$
c2	$\triangleq$	**extends**	c1
		constants	$Connections$
		axioms	$Connections \in SEGMENTS \nrightarrow SEGMENTS$

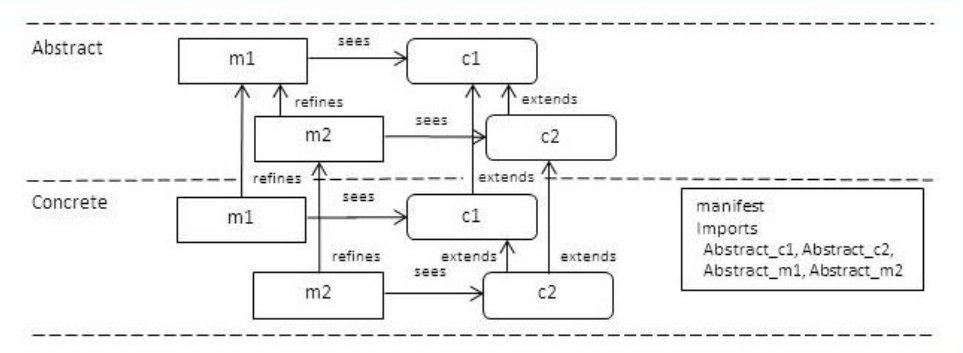

Fig. 4. Assignment of Values to Abstract Model

The contexts c1 and c2 of the concrete model are located in the Concrete project and extends components of the abstract model with concrete values. The machines of the concrete model are identical to the refining machines of the abstract model.

c1	$\triangleq$	**extends**	Abstract_c1
		constants	$s01, s02, s03$
		axioms	partition$(SEGMENTS, \{s01\}, \{s02\}, \{s03\})$
c2	$\triangleq$	**extends**	c1, Abstract_c2
		axioms	$Connections = \{s01 \mapsto s02, s02 \mapsto s03\}$

Since the components of the abstract model are isolated in the Abstract project, it is easy to derive several concrete models of different values and locate them in separate projects for animation and testing.

4 Conclusion

Component sharing is a useful technique in composing a family of Event-B models when variations in models are small. Such models often appears in developing a series of products in industry. The crossed-project reference modularizes shared components by classifying them into hierarchical collections and reducing the risk of unintended change of components. Although component sharing is applicable when the differences between models in a family are simple values assigned to the sets and constants, generic instantiation[6] is more powerful in data instantiation since it enables instantiation of abstract data types in other data types[3]. As generic instantiation uses contexts to represent data types, we expect the crossed-project reference is also helpful for generic instantiation, e.g. in separating data types into library projects. Theory plug-in[4] extends Event-B mathematical language. It allows components to use new data types defined in theories across projects. Since data types and events, represented in contexts and machines respectively, are shared in component sharing, we consider the

crossed-project reference for component sharing is useful in constructing models of components extended with theories similarly.

However, our implementation of the crossed-project reference is experimental and insufficient for practical usage. Most RODIN plug-ins refer the unchecked file of components and are incompatible with our implementation. For example, the components that refer imported components cannot be edited by standard editor tools because they do not recognize imported components. The generic instantiation plug-in also refers unchecked files and does not work with our implementation. Revising our implementation and making it compatible with other modularization plug-ins such as generic instantiation and theory plug-in is future work.

References

1. Abrial, J.-R. : Modeling in Event-B, Cambridge University Press (2010)
2. Abrial, J.-R., Buttler, M., Hallerstede, S., Hoang, T.S., Mehta, F., and Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B, Software Tools for Technology Transfer, 12(6), p.447-466 (2010)
3. Fürst, A., Hoang, T.S., Basin, D., Sato, N., and Miyazaki, K.: Formal system modelling using abstract data types in Event-B, Ameur, Y., Schewe, K.S. (Eds.), ABZ, LNCS 8477, pp. 222-237, Springer (2014)
4. Maamria, I., and Fathabadi, A.S.: Theory Plug-in User Manual, http://wiki.event-b.org/images/Theory_Plugin.pdf (2014)
5. Romanovsky, A., and Thomas, M. (Eds.): Industrial Deployment of System Engineering Methods, Springer (2013)
6. Silva, R., and Butler, M.: Supporting reuse of Event-B developments through generic instantiation, Breitman, K., and Cavalcanti, A. (Eds.), ICFEM, LNCS 5885, pp. 466-484, Springer (2009)

When Students Choose to Use Event-B in their Software Engineering Projects*

J Paul Gibson[1]

SAMOVAR, Télécom Sud Paris, CNRS, Université Paris Saclay,
9 rue Charles Fourier, Evry Cedex, 91011 Paris, France
paul.gibson@telecom-sudparis.eu

Abstract. Students often learn formal methods as part of a software engineering degree programme, without applying these formal methods outside of the specific module(s) dedicated to this subject. In particular, software engineering students often have to build a significant application/program/system in a substantial project at the end of their programme (in order to demonstrate the application of the things they have learned during the previous taught modules). Our experience shows that the majority of students do not use formal methods in this project work. We report on feedback from the minority of students who *did* choose to use formal methods in their projects, and give examples of where this was a help and where it was a hindrance.

Keywords: Teaching , Formal Methods, Technology Transfer

1 Introduction

This paper reports on the continuation of a sequence of publications detailing the author's experience with teaching formal methods. In 1998 [1] reports on the design and implementation of a first (for the authors) formal methods course:

> "Our approach to teaching formal methods tries to give an overall picture rather than concentrating on any one method, language or tool. We believe in letting the students discover the concepts and principles themselves, wherever possible"

Two years later, our approach to teaching fomal methods was integrated into a module dedicated to requirements engineering[2]:

> "Students are encouraged to question the need for formality — each requirements engineering method is a compromise and the use of formal models needs to be placed within the context of the choices that a requirements engineer has to make"

In [3] there is an overview of our approach to weaving formal methods throughout a software engineering programme, using problem based learning, and discussion of the impact of formal methods on the quality of the sofware that the students build:

* This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project http://impex.gforge.inria.fr) from the Agence Nationale de la Recherche (ANR).

> "Anecdotal evidence suggests that the better students adopt formal engineering practices (like the specification of invariants) in projects on other courses which follow their work on the formal methods problems (without being told to do so). Furthermore, the software that these students produce is better than that produced by the other students. However, that should be no surprise as these are the better students!"

In [4], we report on the design of a complete software engineering postgraduate degree programme, where rigour and formality are linked to modelling:

> "All software engineering modules will be taught using a problem-based-learning (PBL) approach. Emphasis will be on rigour and formalilty, and mathematical modelling."

It was at this point in the development of our software engineering program that we decided to use Event-B[5] and the Rodin tool[6] as our 'default' formal method (even though we continued to also use other methods). The decision was based mainly on the positive feedback from various students regarding the RODIN tool, for example:

> "*It was nice to have a formal methods IDE like Eclipse ... you can really experiment with the models and the modelling process ... it makes the maths more like programming ... its the first time I understood the importance of invariants ...* etc."

This paper makes a novel contribution to this sequence of work/publications by reporting on the analysis and feedback (from the students) that we have had since 2011. We are not claiming that this is a scientific study; rather, we report on what we have observed, what the students have stated during feedback interviews and after they have taken up employment after graduating.

The remainder of this paper is structured as follows. Section 2 provides a brief review of relevant related work in the teaching of formal methods. Section 3 motivates the need for the type of study being reported in this paper. Section 4 provides information concerning the students who have participated in this study (through the feedback that they have provided). Section 5 is the main contribution of the paper, where we review the key observations and lessons to be learned. In section 6, we conclude with some recommendtions for teachers of formal methods.

2 Related Work - teaching formal methods

In this section we report on previous work that has had the most influence on our own appraoch to teaching formal methods. It is not intended as a comprehensive review of the history and state-of-the-art.

It is important to note our work is concerned with teaching formal methods to (software) engineering students and not to computer science students[7]. Curriculum design for software engineering students requires making complex trade-offs between the teaching of theory and practice[8, 9]. One of the first books dedicated to the subject of teaching formal method[10] identifies the role

that teachers play in improving the transfer of formal methods technology to industry, through their students. At the turn of the century, a sizeable community of formal methods teachers had grown and started to organise their own workshops concerned with establishing formal methods as a key part of the SoftWare Engineering Body Of Knowledge (SWEBOK)[11]. The need for a "*A Different Software Engineering Text Book*" was identified by Bjorner [12].

The need for scientific evidence supporting the importance of teaching formal methods to software engineers was highlighted by Henderson [13]:

> "Evidence supporting the importance of mathematics in software engineering practice is sparse. This naturally leads to claims that software practitioners dont need to learn or use mathematics. Surveys of current practices reflect reality; many software engineers have not been taught to use discrete mathematics and logic as effective tools. Education is the key to ensuring future software engineers are able to use mathematics and logic as powerful tools for reasoning and thinking."

Before Event-B there was B[14]. Teaching formal methods using B is reported in a number of papers, including: [15–18]

3 Motivation: technology transfer and best practice

The important role of students in the transfer of software engineering technology to industry was illustrated by [19], where the technology in question was UML. Parnas has argued that technology transfer of formal methods will fail because "*We cant sell methods that we dont use ourselves.*"[20]. Our view is a reworking of the phrase from Parnas – our students can't sell formal methods if they don't choose to use them themselves.

Consequently, we wished to observe whether our students choose to use formal methods when working on assessed projects that required the development of software.

4 The Educational Context for our Observations

The MSc program was a 2 year program which ran between 2010 and 2014. The student intake was global from 4 continents — Europe, Africa, Asia and the Americas. Entry to the program was highly selective, with an acceptance rate of between 10 and 20 percent. Subsequently, the number of students in each year was relatively small with an average of 8 per year. As a consequence of the small number of students, our analysis is not based on a scientific (statistically significant) study. Instead, we report on the feedback from students gathered through questionnaires, interviews and informal communication.

5 Observations and Lessons

We structure the observations based on whether the feedback was concerned with project work, placement work or work since graduation.

5.1 Use of formal methods in project work

At the end of the program, the students are expected to work on a significant software engineering project (3-person months per student). They can choose to work in teams or individually. They are free to use whatever techniques/tools/languages/processes that they wish, but they must justify their choice based on the exact nature of the project on which they were working.

After seeing formal methods throughout the program, as well as having a module dedicated to teaching them Event-B and Rodin, we were hoping that the majority of students would write formal specifications in order to model key requirements and/or design issues.

Over the 4 year period, only 3 projects from a total of 14 incorporated significant models in Event-B. These 3 projects were ranked (over the 4 years) in places 1, 3 and 13. The 2 'top' projects were submitted by the best students (based on performance on all modules). They chose to write Event-B models because: " *. . . we wanted to get a better understanding of the rules of the game that we were developing.*", and "*. . . the application was safety critical and we wanted to be sure that the design of the communication protocol was correct.*". For the highest marked project, the team produced a poker game, modelled the rules formally and verified that the operator for ranking hands was based on a transitive relationship. During their development, the RODIN tool helped them identify 'bugs' in their models concerned with misunderstanding of different types of hand. The second project using formal methods developed an android application for use by emergency services when arriving at the scene of an accident. A main issue was how data could be communicated to/from the hospital as effectively as possible. They designed a protocol for the communication but worried that it could lead to deadlocks in the interface. They successfully modelled the protocol in Event-B but were unable to express (or consequently prove) the required property. The project ranked 13 was submitted by a group who chose to use formal methods because they thought that: " *. . . that was what the teacher was looking for.*". They worked on a parallel implementation of a genetic algorithm for pattern recognition. Unfortunately, their lack of experience and ability in formal methods meant that they never finished the specification phase of development, and when they started design and code they were very behind schedule.

Students from projects that chose not to use formal methods were interviewed after they received their evaluations. Two of the groups regretted not writing a formal specification because they had significant problems arising from the team members having inconsistent understanding of their requirements. All groups reported choosing not to use them because they didn't feel that they needed them, and that they wanted to use more agile development approaches (which they felt were not suited to fomal methods).

5.2 Use of formal methods during placement

Through analysis of the student placement reports, and through their presentations, we were able to evaluate the degree of use of formal methods by the students during their placements. We classified the use at 4 different levels:

1. Using formal methods was a critical requirement of the placement (2 students)

2. The student was required to use formal concepts, such as invariants in code, during their placement but there was no dedicated formal methods tool (6 students)
3. The student was not required to use formal methods, butthey were able to use them in their own work. (1 student)
4. The student was not required to use formal methods, and did not use them (20+ students)

It is, perhaps, not surprising that so few students used formal methods during their placements. The 2 students who were obliged to use them had been placed in research and development environments (in education and in industry) where formal methods tools were being developed. The 6 students who were required to use formal concepts were working in safety-critical domains such as the aerospace and health sectors. The one student who chose to try and use formal methods, even though they were not required, reported: "*a certain frustration that my co-workers found it amusing that I would wish to use mathematical models*".

5.3 Use of formal methods after graduating

A significant minority of students(8 in total) stay in regular contact with us after graduating. None of them are working in an environment which uses formal methods. A handful of them believe that the quality of their work would be improved through the use of formal methods.

6 Conclusions: recommendations for teachers

Although our report is based on a small number of observations, it is worrying that Parnas appears to be (at least partially) correct when he stated that we will not be able to transfer formal methods technology from academia to industry.

It is not the teachers' role to force their students to use formal methods. Successful teaching of formal methods will motivate students to use them because they believe in them. We, as teachers, need to better monitor students during the whole of their academic careers (and after) to measure the use of formal methods, together with the impact of their use on the quality of software being developed. We also need to better support students who wish to introduce formal methods technologies in their workplace.

References

1. Gibson, J., Méry, D.: Teaching formal methods: Lessons to learn. In Flynn, S., Butterfield, A., eds.: 2nd Irish Workshop on Formal Methods (IWFM 1998). Electronic Workshops in Computing, Cork, Ireland, BCS (July 1998)
2. Gibson, J.: Formal requirements engineering: Learning from the students. In Grant, D., ed.: 12th Australian Software Engineering Conference (ASWEC 2000), Canberra, Australia, IEEE Computer Society (2000) 171–180
3. Gibson, J.: Weaving a formal methods education with problem-based learning. In Margaria, T., Steffen, B., eds.: 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Volume 17 of Communications in Computer and Information Science (CCIS)., Porto Sani, Greece, Springer-Verlag, Berlin Heidelberg (October 2008) 460–472

4. Gibson, J., Raffy, J.L.: A future-proof postgraduate software engineering programme: Maintainability issues. In: The Sixth International Conference on Software Engineering Advances(ICSEA 11), Barcelona, Spain (October 2011) 471–476
5. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
6. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-b. Int. J. Softw. Tools Technol. Transf. **12** (November 2010) 447–466
7. Parnas, D.L.: Software engineering programs are not computer science programs. Software, IEEE **16**(6) (1999) 19–30
8. Garlan, D.: Formal methods for software engineers: Tradeoffs in curriculum design. In: Software Engineering Education. Springer (1992) 131–142
9. Garlan, D.: Making formal methods education effective for professional software engineers. Information and Software Technology **37**(5) (1995) 261–268
10. Dean, N., Hinchey, M.: Teaching and learning formal methods. Morgan Kaufmann (1996)
11. Almstrum, V.L., Dean, C.N., Goelman, D., Hilburn, T.B., Smith, J.: Support for teaching formal methods. SIGCSE Bull. **33**(2) (2001) 71–88
12. Bjørner, D.: On teaching software engineering based on formal techniques - thoughts about and plans for - a different software engineering text book. J. UCS **7**(8) (2001) 641–667
13. Henderson, P.B.: Mathematical reasoning in software engineering education. Communications of the ACM **46**(9) (2003) 45–50
14. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (1996)
15. Leuschel, M., Samia, M., Bendisposto, J., Luo, L.: Easy graphical animation and formula visualisation for teaching b. In: Formal Methods in Computer Science Education (FORMED). (March 2008)
16. Habrias, H.: Teaching specifications, hands on. In: Formal Methods in Computer Science Education (FORMED). (March 2008) 5–15
17. Mry, D.: Teaching programming methodology using event b. In Habrias, H., ed.: The B Method: from Research to Teaching. (June 2008)
18. Guyomard, M., Alain, P., Hadjali, A., Jaudoin, H., Smits, G.: First balance sheet of a formal approach in the teaching of data structures. The B method: from Research to Teaching 66–91
19. Hallinan, S., Gibson, J.: A graduate's role in technology transfer: From requirements to design with UML. In Kokol, P., ed.: IASTED International Conference on Software Engineering, part of the 23rd Multi-Conference on Applied Informatics, Innsbruck, Austria, IASTED/ACTA Press (2005) 94–99
20. Parnas, D.L.: "formal methods" technology transfer will fail. Journal of Systems and Software **40**(3) (1998) 195–198

Automating Verification of Event-B Models

Paulius Stankaitis, Alexei Iliasov,
David Adjepon-Yamoah, Alexander Romanovsky

Centre for Software Reliability,
Newcastle University,
Newcastle upon Tyne, UK

{paulius.stankaitis, alexei.iliasov, d.e.adjepon-yamoah,
alexander.romanovsky}@ncl.ac.uk

Abstract. Event-B is one of more popular notations for model-based, proof driven specification. It offers a fairly high-level mathematical language based on FOL and ZF set theory and an economical yet expressive modelling notation. Model correctness is established by discharging proving a number conjectures constructed via a syntactic instantiation of schematic conditions. A large proportion of provable conjectures requires proof hints from a user. For larger models this becomes extremely onerous as identical or similar proofs have to be repeated over and over, especially after model refactoring stages. In the paper we briefly present a new Rodin Platform proof back-end based on the Why3 umbrella prover.

1 Introduction

Event-B [1] is one of more popular notations for a model-based, proof driven specification. It offers a fairly high-level mathematical language based on FOL and ZF set theory and an economical yet expressive modelling notation centred around the notion of an atomic event - a form of before-after predicate. Leaving aside the methodological qualities of Event-B, one can regard an Event-B model as a high-level notation from which a number of *proof obligations* may be automatically derived. Proof obligations seek to establish properties like the preservation of invariant and satisfaction of refinement obligations. A model is deemed correct when all proof obligations are successfully discharged.

Recently some important work has been done to bring a large number of TPTP and SMT-LIB provers under the roof of a common, versatile notation - the Why3 verification platform. At the basic level Why3 offers a common interface to over a dozen of automated provers; it also has its own high-level specification notation to reason about software correctness though we do not make any use of it in this work and rather rely on Why3 to offer a bridge to tools like Z3, SPASS, Vampire and Alt-Ergo.

A theorem prover is a computationally and memory intensive program typically run for rather short periods of time (the vast majority of proofs is done within two seconds) with long idling periods in between. Proof success and perceived usability depend on the capability of an execution platform. Such requirement is best met by the cloud technology.

Doing proofs on a cloud opens possibilities that we believe were previously not explored, outside, perhaps, prover contests. The cloud service keeps a detailed record of each proof attempt along with (possibly obfuscated) proof obligations, supporting lemmas and translation rules. There is a fairly extensive library of Event-B models constructed over the past 15 years and these are a ready of source

2 Rodin Why3 plug-in

Development in Event-B is supported by the Rodin Platform [5] that has been under active development since 2005. It has been long recognised that the Rodin Platform may significantly benefit from an interface between Event-B and TPTP [11] provers. To simplify translation we decided to use the Why3 [2] umbrella prover that offers a single and quite palatable input notation and also supports SMT-LIB compliant provers. Why3 supports 16 external automatic provers (not counting different versions of the same tool), these include all the state-of-the-art tools like Z3 [10], SPASS [12], Vampire [9] and Alt-Ergo [4].

A plug-in to the Rodin Platform was realised [8] to map between the Event-B mathematical language and the Why3 *theory* input notation (we do not make use of its other part - a modelling language notation). The syntactic part of the translation is trivial: just one Tom/Java class mapping between Event-B and Why3 operators. The bulk of the effort is in the axioms and lemmas defining the properties of the numerous Event-B set-theoretic constructs. We have a working prototype able to discharge (via provers like SPASS and Alt-Ergo) a number of properties that previously required interactive proof. At the same time, we realise that axiomatisation of a complex mathematical language like the one of Event-B is likely to be an ever open problem. It is apparent that different provers prefer differing styles of operator definitions: some perform better with an inductive style (i.e., to define set cardinality one may say that the size of an empty set is zero, adding one element to a set increases its size by one) while others prefer regress to already known concepts (there exists a bijection such that ...). Since we do not know how to define one best axiomatization, even for any one given prover, we offer an open translator with which a user may define, with as many cross-checks as practically reasonable, a custom embedding of Event-B into the Why3.

3 Generic lemmata

Throughout our research we discovered that with the new verification tool we could address another interactive proof problem - fragility and non reusability. There is a number of circumstances when existing interactive proofs become invalidated and a new version of an undischarged proof obligation appears.

On rare occasions a model or its sizeable part are changed significantly so that there is no or little connection between old and new proof obligations. Far more common are incremental changes that alter the goal, set of hypotheses,

identifier names or types. During the refactoring of a refinement tree it is very common to lose a large proportion of manual proofs.

While there is a potential to improve the way the Rodin Platform handles interactive proofs, the fragility of such proofs has mainly to do with their nature. Unlike more traditional theorems and lemmas found in maths textbooks, model proof obligations have no meaning outside of the very narrow model context. And since Event-B relies on syntactic proof rules for invariant and refinement checks, even fairly superficial syntactic changes would result in new proof obligations which are, in fact, if not logically equivalent are often quite similar to the deleted ones.

Even in the case of a significant model change, it is, in our experience, likely that proof obligations similar to those requiring an interactive proof re-appear. In addition, there is a large number of essentially identical interactive proofs re-appearing in different projects due to specific weaknesses in the underlying automatic provers.

The key to our approach is understanding what 'similar' means in the relation to some two proof obligations. One interpretation is that similar conditions can be discharged by the same proof scripts. To make it practical, this has to be relaxed with some form of a proof script template [6]. The interpretation we take in this work is that two proof obligations are similar if they both can be discharged by adding same schematic lemma to the set of their hypotheses. This definition is rather intricately linked with the capabilities of underlying automated provers: adding a tautology (a proven lemma) to hypotheses does not change a conjecture but it might help to guide an automated prover to successful proof completion.

It is our experience that the existing the Rodin automatic provers do not benefit from adding a schematic lemma (with instantiated type variables, to make it first order) to hypotheses and they still need to be instantiated manually by manually by an engineer to have any effect. However, in the case of the Why3 plug-in, with which this approach has a close integration, it is different: a fitting schematic lemma in hypotheses makes proof nearly instantaneous.

There are situations when the only viable way to complete a proof is by providing a proof hint. One such case - refinement of event parameters - is adequately addressed at the modelling notation level where a user is requested to provide a witness as a part of a specification. There are proposals to generalise this, for the majority of situations, and define hints at the model level [7].

A schematic lemma considered on its own is of a little use. But if a proof obligation can be proven by adding a schematic lemma, then the construction of a schematic lemma in itself a proof process. As a simple illustration, consider the following (trivial) conjecture:

$$
\begin{array}{l}
\mathit{library} \in \mathrm{BOOKS} \to \mathbb{N} \\
b \in \mathrm{BOOKS} \land c \in \mathbb{N} \\
\ldots \\
\vdash \\
\mathit{library} \lhd\!\!+ \{b \mapsto c\} \in \mathrm{BOOKS} \to \mathbb{N}
\end{array}
$$

And suppose there were no automated prover capable of discharge it. It is clear that the crux of the statement is in the interaction of functional override, totality and functionality. The above can be rewritten as

$$
\begin{array}{l}
f \in A \to B \\
\vdash \\
\forall x, y \cdot x \in A \land y \in B \Rightarrow f \lhd\!\!+ \{x \mapsto y\} \in A \to B
\end{array}
$$

Since the Event-B mathematical language does not have type variables such a condition may only be defined either for specific A's and B's, or, in a slightly altered form, using the Theory plug-in [3]. But to discharge the original proof obligation one still needs to find this lemma and instantiates it. It is a tedious and error-prone process for a human but a fairly trivial task for a certain kind of automated provers.

The example above is quite generic in the sense it is potentially useful for in many other contexts. At times a schematic lemma need to be fairly concrete. It is also easier to write a lemma that narrowly targets a proof obligation. This distinction between 'general' and 'specific' is, at the moment, completely subjective and relies on the modeller's intuition. To reflect the fact that a more general lemma is more likely to be reused, schematic lemmas are classified into three visibility classes: machine (single model), project (collection of models) and global. A machine-level lemma will be considered for a proof obligation of the machine with which the lemma is associated; similarly, for the project-level attachment. A global schematic lemma becomes a part of the Event-B mathematical language definition for the Why3 plug-in.

Just as model construction is often an iterative process, we have discovered during our experiments that finding a good schematic lemma may require several attempts. A common scenario is that an existing lemma may be relaxed so that while it is still strong enough to discharge conditions that were dependent on it, it can also discharge some new ones. For instance, we have seen several cases where a fairly narrow and detailed lemma would gradually slim down to a simple (and much more valuable) statement about distributivity of certain operators. It does require at times a considerable effort to come up with an abstract and minimal covering condition but the result is rewarding and reusable across projects.

4 Hypotheses and lemmata filtering

The initial experiments have shown that a minimal axiomatisation support is not sufficient to discharge a sizeable proportion of proof obligations. Provable lemmas were added to assist with specific cases but then it become clear that a large number of support conditions slow down or even preclude a proof. On top of that, the auto tactic language of Rodin offers a very crude hypotheses selection mechanism that for larger models tends to include tens if not hundreds of irrelevant statements. It was thus deemed essential to attempt to filter out unnecessary axiomatisation definitions, Why3 support lemmas and proof obligation hypotheses.

The Rodin mechanism for hypotheses filtering is based on matching conditions with common free identifiers. To complement this mechanism we do filtering on the structure of a formula. It is also a natural choice since support lemmas do not have any free identifiers.

Directly comparing some two formulae is expensive: a straightforward algorithm (tree matching) is quadratic unless memory is not an issue. We use a computationally cheap proxy measure known as the Jaccard similarity which, as the first approximation, is defined as:

$$JS(P,Q) = \operatorname{card}(P \cap Q) / \operatorname{card}(P \cup Q) \tag{1}$$

The key is in computing the number of overall and common elements and, in fact, defining what an "element" means for a formula. One immediate issue is that P and Q are sets and a formula, at a syntactic level, is a tree.

One common way to match some two sequences (e.g., bits of text) using the Jaccard similarity is to use *shingles* of elements to attempt to capture some part of the ordering information. A shingle is a tuple preserving order of original elements but seen as an atomic element. Thus sequence $[a,b,c,d]$ could be characterised by two 3-shingles $P = \{[a,b,c],[b,c,d]\}$ (here $[b,c,d]$ is but a structured name) and matching based on these shingles would correctly show that $[a,b,c,d]$ is much closer to $[a,b,c,d,e]$ than to $[d,c,b,a]$. Trees are slightly more challenging.

On one hand, a tree may be seen (but not defined uniquely) as a set of paths from a root to leaves and we could just do matching on a set of sequences and aggregate the result. This is not completely satisfactory as tree structure is not accounted for. So we add another characterisation of tree as a set of sequences of the form $[p, c_1, \ldots, c_2]$ where p is a parent element and $c_1, \ldots, c_2$ are children. This immediately gives a set of n-shingles that might need to be converted into shorter m-shingles to make things practical.

As an example, consider the following expression $a * (b + c/d) + e * (f - d * 2)$. We are not interested in identifiers and literals so we remove them to obtain tree $+(*(+/))(*(-*))$ which has the following 3-shingles based on paths, $[*,+,/],[+,*,+],[+,*,-],[*,-,*]$, and only 1 3-shingle, $[+,*,*]$, based on the structure. The shingles are quite cheap to compute (linear to formula size) and match (fixed cost if we disregard low weight shingles, see below). Let $\mathrm{sd}(P)$ and $\mathrm{sw}(P)$ be set of depth and structure shingles of formula P. Then the similarity between some P and Q is computed as:

$$\begin{aligned} s(P,Q) &= \textstyle\sum_{i \in I_1} \mathrm{w_d}(i) + c \sum_{i \in I_2} \mathrm{w_w}(i) \\ I_1 &= \mathrm{sd}(P) \cap \mathrm{sd}(Q), I_2 = \mathrm{sw}(P) \cap \mathrm{sw}(Q) \end{aligned} \tag{2}$$

where $w_*(i) = \mathrm{cnt}(i)^{-1}$ and $\mathrm{cnt}(i)$ is number of times i occurs in all hypotheses and support lemmas. Very common shingles contribute little to the

similarity assessment and may be disregarded so that there is some k such that $\text{card}(I_1) < k, \text{card}(I_2) < k$.

References

1. Jean-Raymond Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
2. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, August 2011.
3. Michael Butler and Issam Maamria. *Practical Theory Extension in Event-B*, pages 67–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
4. Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.
5. DEPLOY. Event B and the Rodin Platform. http://www.event-b.org/.
6. Leo Freitas and Iain Whiteside. Proof Patterns for Formal Methods. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 279–295, 2014.
7. Thai Son Hoang. Proof Hints for Event-B. *CoRR*, abs/1211.1172, 2012.
8. Alexei Iliasov, Paulius Stankaitis, David Adjepon-Yamoah, and Alexander Romanovsky. Rodin platform why3 plug-in. *To appear in the proceedings of ABZ2016*, 2016.
9. Laura Kovács and Andrey Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, 2013.
10. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08. Springer, 2008.
11. TPTP. Thousands of Problems for Theorem Provers. Available at www.tptp.org/.
12. Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22 : 22nd International Conference on Automated Deduction*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

A Formal Approach to Use Case Driven Testing

Rajiv Murali, Andrew Ireland, and Gudmund Grov
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, UK

Abstract. UML use cases are a popular technique used to define and communicate the behavioural requirements for software-intensive systems. They appear in two complementary forms: (1) a *use case diagram* that provides an easy to understand illustration of use case modelling elements, i.e. use cases, actors, dependency relationships, etc.; and (2) an accompanying *textual use case specification* that details the behaviour and constraints for each use case. Traditionally, the use case specifications are documented informally outside the UML model, often in a text document, with no structure or traceability to other UML modelling elements, e.g. actors, subject, dependency relationships, etc. This often results in inconsistencies between the UML model and the use case specification, and introduces a barrier to more automated methods for analysis. We describe an approach that extends the UML model to enable use cases to contain a structured use case specification. This extension is then used to support methods to: (1) generate activity diagrams to visualise the behaviour of the use cases; and (2) include our previous work on formalising the use case specification with Event-B to support the generation of test cases. An example of an Anti-lock Braking System (ABS) is used to describe our approach. An implementation of this approach is provided via a plug-in, *UsecasePro*, for the UML modelling tool Papyrus.